LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-TR-214714

Distributed Breadth-First Search with 2-D Partitioning

Edmond Chow, Keith Henderson, Andy Yoo

August 19, 2005

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Distributed Breadth-First Search with 2-D Partitioning*

Edmond Chow, Andy Yoo, Keith Henderson
Lawrence Livermore National Laboratory

Abstract

Many emerging large-scale data science applications require searching large graphs distributed across multiple memories and processors. This paper presents a scalable implementation of distributed breadth-first search (BFS) which has been applied to graphs with over three billion vertices. The main contributions of this paper are the comparisons of a 2-D (edge) partitioning of the graph to the more common 1-D (vertex) partitioning and the scalable implementation of the algorithm. For Poisson random graphs which have low diameter like many realistic information network data, we determine when one type of partitioning is advantageous over the other. Also for Poisson random graphs, we show that memory use is scalable. We have demonstrated the scalability of our algorithm on a latest large-scale parallel computer, IBM BlueGene/L, by searching a very large graph with more than three billion vertices. It is shown that our algorithm is highly memory-intensive and the timing is related to the number of synchronization steps in the algorithm.

1 Introduction

Many data science applications are beginning to use sparse relational data in the form of graphs. For example, to determine the nature of the relationship between two vertices in a relational data graph, the shortest graph distance paths between the two vertices can be used [1, 7]. Breadth-first search (BFS) may be applied for this task. More efficient algorithms based on bi-directional or heuristic search are also possible but fundamentally have the same structure as BFS. Indeed, BFS is used in a myriad of applications and is related to sparse matrices and sparse matrix-sparse vector multiplication. Distributed BFS (and shortest

*LLNL Technical Report UCRL-CONF-210829.

path) algorithms are not new. There is in particular much literature on the case where each processing element stores a single vertex in the graph. See, for example, [17] for a review of results.

In this paper, we study the behavior of BFS on very large-scale graphs partitioned across a large distributed-memory supercomputer. A critical question is how to partition the graph, which will affect the communication and overall time for BFS. This paper is novel in that it studies distributed BFS with a 2-D partitioning of the graph, corresponding to a partitioning of the edges. The more common 1-D partitioning corresponds to a partitioning of the vertices. 1-D partitioning is simpler and leads to low communication volume and number of messages for graphs arising in many scientific applications, such as the numerical solution of partial differential equations (PDEs). Many graphs arising in data science applications, however, have properties very different from PDE meshes. In particular, these graphs have low average path length. That is, the typical length of the shortest distance path between any two vertices is small. For example, the average path length may vary as $O(\log n)$ where there are n vertices in the graph. 1-D partitioning for graphs with this property leads to high communication volume and total number of messages.

We thus consider 2-D partitioning. Besides potentially lower communication volume, the advantage of 2-D partitioning over 1-D partitioning is that collective communications involve only $O(\sqrt{P})$ processors rather than P processors in the 1-D case. 2-D partitioning is most well-known for dense matrix computations [8] and has been proposed for sparse matrix computations where the structure of the matrix is difficult to exploit [10]. However, 2-D partitioning is rarely used, perhaps because most problems to date have sufficient structure so that 1-D partitioning is adequate. Further, whereas there are many algorithms and codes for generating 1-D partitionings by optimizing communication volume [5, 9, 11], algorithms and codes for 2-D partitioning have only recently become available [4, 6].

There are several variants of 2-D partitioning. The most useful variant has been called *transpose free doubling/halving* [13], *redistribution-free method* [12], and *2-D checkerboard* [6]. In this paper, we will simply call this 2-D partitioning, and will describe it in the next section. For other variants, see [4, 8, 10, 12, 13, 16].

In this paper, we study Poisson random graphs, where the probability of an edge connecting any two vertices is constant. We have used the Poisson random graphs, because there are no publicly available large real graphs with which we can test the scalability of the proposed BFS algorithm. A social network graph derived from the World Wide Web, for example, contains 15 million vertices [7] and the largest citation network available has two million vertices [15]. In the absence of the large real graphs, the synthetic random graphs, which are the simplest graphs that have small diameter, the feature of real-world networks, provide us with an easy means to construct very large graphs with *billions* of vertices. Furthermore, the random graphs have almost no clustering and thus have large edge-cuts when partitioned, allowing us to understand the worst-case performance of our algorithm.

We use arbitrary partitionings with the constraint that the partitions are balanced in terms of number of vertices and edges. For P processors and n vertices, we define a scalable implementation to be one that uses $O(n/P)$ storage per processor. For Poisson random graphs, we will show in particular that the sum of the sizes of the messages sent by one processor to other processors is $O(n/P)$ for fixed average degree k. This result applies to both 1-D and 2-D partitionings.

This paper is organized as follows. In Section 2, we discuss implementations of distributed BFS for 1-D and 2-D partitioning. In particular, there are several choices for the communication operations. In Section 3, for a Poisson random graph, we discuss the size of the messages in these communication operations in terms of parameters of the graph and the partitioning. In Section 4, we show the performance results of parallel BFS in the 1-D and 2-D cases. Section 5 concludes the paper with several directions for future research.

2 Proposed Distributed BFS Algorithm

In this section, we present the implementation of distributed BFS with 1-D and 2-D partitioning. The 1-D algorithm is a special case of the 2-D algorithm. The algorithms are described in the next two subsections. Then we discuss two important issues: options for the communication steps, and scalable data structures.

In the following, we use P to denote the number of processors, n to denote the number of vertices in a Poisson random graph, and k to denote the average degree of the graph. We will consider undirected graphs only in this paper. We define the *level* of a vertex as its graph distance from a start vertex. Level-synchronized BFS algorithms proceed level-by-level starting with the start vertex (or a set of start vertices). We define the *frontier* as the set of vertices in the current level of the BFS algorithm.

2.1 Distributed BFS with 1-D Partitioning

A 1-D partitioning of a graph is a partitioning of its vertices such that each vertex and the edges emanating from it are owned by one processor. The set of vertices owned by a processor is also called its *local vertices*. The following illustrates a 1-D P-way partitioning using the adjacency matrix, A, of the graph, symmetrically reordered so that vertices owned by the same processor are contiguous.

$$\left[\begin{array}{c} A_1 \\ \hline A_2 \\ \hline \vdots \\ \hline A_P \end{array}\right]$$

The subscripts indicate the index of the processor owning the data. The edges emanating from vertex v form its *edge list* and is the list of vertex indices in row v of the adjacency matrix. For the partitioning to be balanced, each processor should be assigned approximately the same number of vertices and emanating edges.

A distributed BFS with 1-D partitioning proceeds as follows. At each level, each processor has a set F which is the set of frontier vertices owned by that processor. The edge lists of the vertices in F are merged to form a set N of neighboring vertices. Some of these vertices will be owned by the same processor, and some will be owned by other processors. For vertices in the latter case, messages are sent to other processors (neighbor vertices are sent to their owners) to potentially add these vertices to their frontier set for the next level. Each processor receives these sets of neighbor vertices and merges them to form a set $\bar{N}$ of vertices which the processor owns. The processor may have marked some vertices in $\bar{N}$ in a previous iteration. In that case, the processor will ignore this message, and all subsequent messages regarding those vertices.

Algorithm 1 describes distributed breadth-first expansion using a 1-D partitioning, starting with a vertex v_s. In the algorithm, every vertex v becomes labeled with its level, $L_{v_s}(v)$, i.e., its graph distance from v_s. (BFS is a simple modification of breadth-first expansion.) The data structure $L_{v_s}(v)$ is also distributed so that a processor only stores L for its local vertices.

2.2 Distributed BFS with 2-D partitioning

A 2-D (checkerboard) partitioning of a graph is a partitioning of its edges such that each edge is owned by one processor. In addition, the vertices are also partitioned such that each vertex is owned by one processor, like in 1-D partitioning. A process stores some edges incident on its owned vertices, and some edges that are not. This partitioning can be illustrated using the adjacency matrix, A, of the graph, symmetrically reordered so that vertices owned by the same processor are contiguous.

$$
\left[\begin{array}{c|c|c|c}
A^{(1)}_{1,1} & A^{(1)}_{1,2} & \cdots & A^{(1)}_{1,C} \\ \hline
A^{(1)}_{2,1} & A^{(1)}_{2,2} & \cdots & A^{(1)}_{2,C} \\ \hline
\vdots & \vdots & \ddots & \vdots \\ \hline
A^{(1)}_{R,1} & A^{(1)}_{R,2} & \cdots & A^{(1)}_{R,C} \\ \hline
\multicolumn{4}{c}{\vdots} \\
\multicolumn{4}{c}{\vdots} \\
\multicolumn{4}{c}{\vdots} \\ \hline
A^{(C)}_{1,1} & A^{(C)}_{1,2} & \cdots & A^{(C)}_{1,C} \\ \hline
A^{(C)}_{2,1} & A^{(C)}_{2,2} & \cdots & A^{(C)}_{2,C} \\ \hline
\vdots & \vdots & \ddots & \vdots \\ \hline
A^{(C)}_{R,1} & A^{(C)}_{R,2} & \cdots & A^{(C)}_{R,C}
\end{array}\right]
$$

Here, the partitioning is for $P = RC$ processors, logically arranged in a $R \times C$ processor mesh. We will use the terms *processor-row* and *processor-column* with respect to this processor mesh. In the 2-D partitioning above, the adjacency matrix is divided into RC block rows and C block columns. The notation $A^{(*)}_{i,j}$ denotes a block owned by processor (i, j). Each processor owns C blocks. To partition the vertices, processor (i, j) owns the vertices corresponding to block row $(j - 1)R + i$. For the partitioning to be balanced, each processor should be assigned approximately the same number of vertices and edges. The 1-D case is equivalent to the 2-D case with $R = 1$ or $C = 1$.

For 2-D partitioning, we assume that the edge list for a given vertex is a *column* of the adjacency matrix, which will slightly simplify the description of the algorithm. Thus each block contains *partial* edge lists. In BFS using this partitioning, each processor has a set F which is the set of frontier vertices owned by that processor. Consider a vertex v in F. The owner of v sends messages to other processors in its processor-column to tell them that v is on the frontier, since any of these processors may contain partial edge lists for v. The partial edge lists on each processor are merged to form the set N, which are potential vertices on the next frontier. The vertices in N are then sent to their owners to potentially be added to the new frontier set on those processors. With 2-D partitioning, these owner processors are in the same processor row. The advantage of 2-D partitioning over 1-D partitioning is that the processor-column and processor-row communications involve R and C processors, respectively; for 1-D partitioning, all P processors are involved in the communication operation.

Algorithm 2 describes distributed breadth-first expansion using a 2-D partitioning, starting with a vertex v_s. In the algorithm, every vertex v becomes labeled with its level $L_{v_s}(v)$, i.e., its graph distance from v_s.

2.3 Communication Options

In this subsection, we describe the communication operations of the above algorithms in detail. Algorithm 1 for 1-D partitioning has a single communication step, in lines 8–13. Algorithm 2 for 2-D partitioning, in contrast, has two communication steps: lines 7–11, called *expand* communication, and lines 13–18, called *fold* communication. The communication step in the 1-D algorithm is the same as the fold communication in the 2-D algorithm.

2.3.1 Expand communication

In the expand operation, processors send the indices of the frontier vertices that they own to other processors. For dense matrices [8] (and even in some cases for sparse matrices [10]), this operation is traditionally implemented with an all-gather collective communication, since all indices owned by a processor need to be sent. For BFS, this is equivalent to the case where all vertices are on the frontier. This communication is not scalable as the number of processors increases.

For sparse graphs, however, is it advantageous to only send vertices on the frontier, and to only send to processors that have non-empty partial edge lists corresponding to these frontier vertices. This operation would now be implemented by an all-to-all collective communication. In the 2-D case, each processor needs to store information about the edge lists of other processors in its processor-column. The storage for this information is proportional to the number of vertices owned by a processor, and hence it is scalable. We will show in Section 3 that for Poisson random graphs, the message lengths are scalable when communication is performed this way.

2.3.2 Fold communication

The fold communication is traditionally implemented for dense matrix computations as an all-to-all operation. We will show in Section 3 that the message lengths are scalable in this operation for Poisson random graphs.

An alternative is to implement the fold communication as a reduce-scatter operation. We have implemented the fold communication based on the reduce-scatter operation and have shown that this can significantly reduce the total communication volume in Section 4. In this case, in Algorithm 2, each processor receives its $\bar{N}$ directly and line 18 of the algorithm is not required. The reduction operation occurs within the reduction stage of the operation and eliminates duplicate vertex indices being sent to a processor. This reduction operator is not provided by MPI, and thus this reduce-scatter operation must be coded using sends and receives.

2.4 Scalable data structures and optimizations

Storage of edge lists

In the 2-D case, each processor owns $O(n/P)$ vertices but contains edge lists for $O(n/C)$ vertices, which is asymptotically larger than $O(n/P)$. For Poisson random graphs, however, the expected number of non-empty edge lists is $O(n/P)$. This can be demonstrated by examining the case where P is large.

Each edge list has on average k entries, and these are distributed among R processes randomly. As $R \to \infty$, it is increasingly unlikely that two of these entries will be in a given edge list. In the limit, then, the length of every edge list is either zero or one, and there is one edge list per edge. The total number of edges in the graph is nk, so we expect nk non-empty edge lists distributed over P processors.

When P is smaller some edge lists will be longer, so there are fewer non-empty edge lists in that case. An upper bound on the number of edge lists in the graph is nk, so the expected number of non-empty edge lists on a given processor is $O(n/P)$. Thus it is necessary not to index all edge lists, but only the non-empty ones.

Similarly, the number of unique vertices appearing in all edge lists on a processor is $O(n/P)$. This is demonstrated by noting that we could store edge lists for matrix rows rather than columns. The above analysis applies here, so the number of non-empty row edge lists would be $O(n/P)$ as well. Each non-empty row edge list on a processor corresponds to a unique entry in the column edge lists on that processor.

Local indexing

The index of a vertex in the original numbering of the graph vertices is called the *global* index of the vertex. To facilitate the storage of data associated with local vertices (e.g., L), the global index of a local vertex is mapped to a *local* index. This local index is used to access L and other arrays. The mapping from global indices to local indices is accomplished with a hash table.

Each processor stores three such mappings. The vertices owned by a processor are indexed locally, as noted above. Moreover, a mapping is generated for any vertices with non-empty edge lists on a processor. Lastly, a mapping is generated for any vertices appearing in an edge list on a processor. As described previously, the latter two mappings include $O(n/P)$ vertices each. The first mapping is obviously $O(n/P)$.

Sent neighbors

Each processor keeps track of which neighbor vertices it has already sent. Once a neighbor vertex is sent, it may be encountered again, but it never needs to be sent again. The storage required for this optimization is proportional to the number of unique vertices that appear in edge lists on a given processor, which is $O(n/P)$ as demonstrated above.

3 Message Lengths for Random Graphs

For Poisson random graphs, we can derive bounds on the length of the communication messages in distributed BFS. Recall that we define n as the number of vertices in the graph, k as the average degree, and P as the number of processors. We assume P can be factored as $P = R \times C$, the dimensions of the processor mesh in the 2-D case. For simplicity, we further assume that n is a multiple of P and that each processor owns n/P vertices.

Let A' be the matrix formed by any m rows of the adjacency matrix of the random graph. We define the useful quantity

$$\gamma(m) = 1 - \left(\frac{n-1}{n}\right)^{mk}$$

which is the probability that a given column of A' is nonzero. The quantity mk is the expected number of edges (non-zeros) in A'. The function γ approaches mk/n for large n and approaches 1 for small n.

For distributed BFS with 1-D partitioning, processor i owns the A_i part of the adjacency matrix. In the communication operation, processor i sends the indices of the neighbors of its frontier vertices to their owner processors. If all vertices owned by i are on the frontier, the expected number of neighbor vertices is

$$n \cdot \gamma(n/P) \cdot (P-1)/P.$$

This communication length is nk/P in the worst case, which is $O(n/P)$. The worst case viewed another way is equal to the actual number of non-zeros in A_i; every edge causes a communication. This worst case result is independent of the graph.

In 2-D expand communication, the indices of the vertices on the frontier set are sent to the $R-1$ other processors in the processor-column. In the worst case, if all n/P vertices owned by a processor are on the frontier (or if all-gather communication is used and all n/P indices are sent) the number of indices sent by the processor is

$$\frac{n}{P}(R-1)$$

which increases with R and thus the message size is not controlled when the number of processors increases.

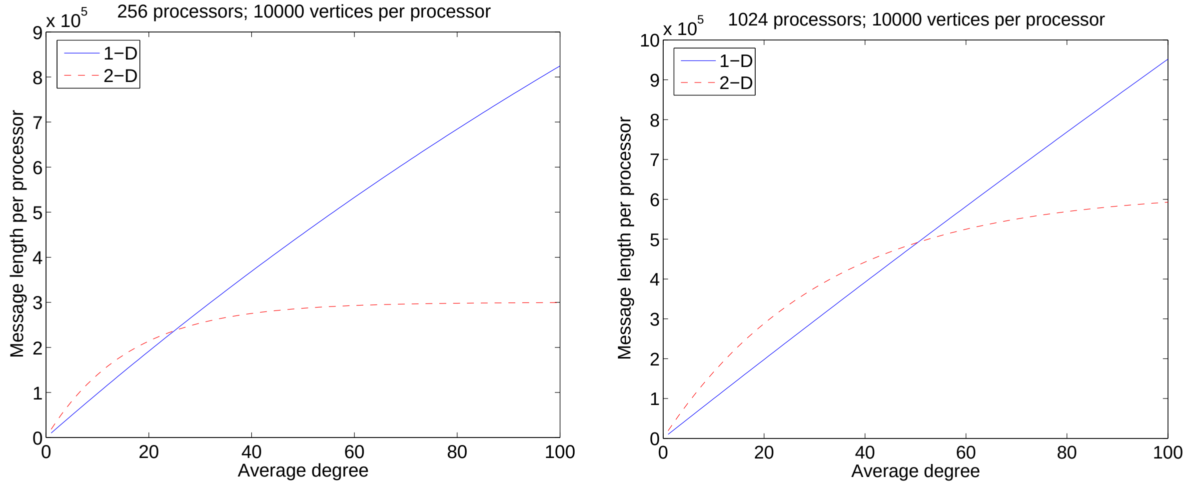
The maximum expected message size is bounded as R increases, however, if a processor only sends the indices needed by another processor (all-to-all communication, but requires knowing which indices to send). A processor only sends indices to processors that have partial edge lists corresponding to vertices owned by it. The expected number of indices is

$$\frac{n}{P} \cdot \gamma(n/R) \cdot (R-1).$$

The result for the 2-D fold communication is similar:

$$\frac{n}{P} \cdot \gamma(n/C) \cdot (C-1).$$

These two quantities are also $O(n/P)$ in the worst case. Thus, for both 1-D and 2-D



(a) 256 processors (b) 1024 processors

Figure 1: Message lengths for 256 and 1024 processors.

partitioning, the length of the communication from a single processor is $O(n/P)$, proportional to the number of vertices owned by a processor.

For illustration purposes, Figure 1 shows the message lengths for a single processor for 1-D and 2-D partitioning (sum of expand and fold communication in the 2-D case) as a function of the average degree, k. The results show that 2-D partitioning is advantageous when average degree is large. When more processors are used, the cross-over point is at a higher k.

4 Performance Results

This section presents experimental results for the distributed BFS for Poisson random graphs with 1-D and 2-D partitionings. We have conducted the experiments on the IBM Blue-Gene/L [2] and MCR [14], a large Linux cluster with a Quadrics switch, both located at Lawrence Livermore National Laboratory. In the experiments, 100 pairs of start and target

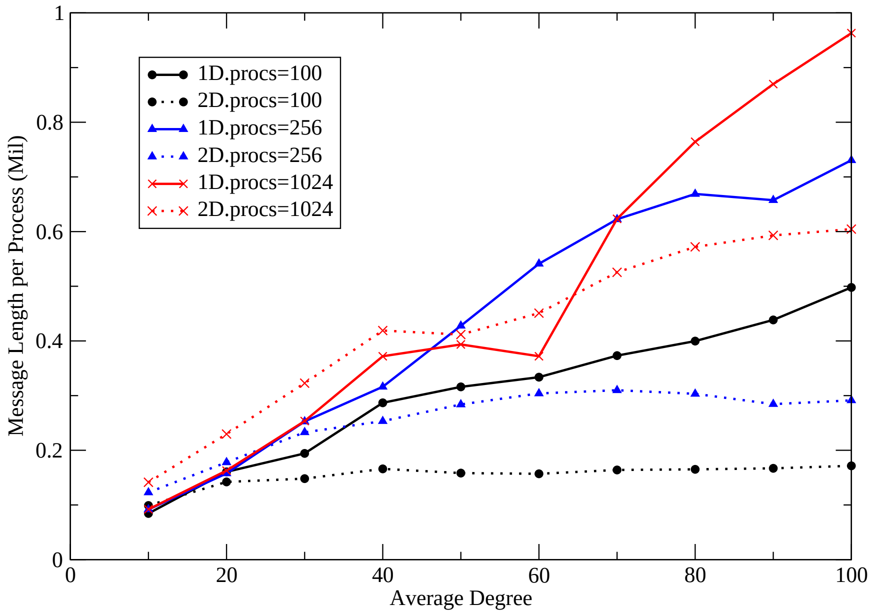


Figure 2: Effect of average degree on message lengths, 10000 vertices per processor.

vertices are selected randomly, and the timings reported are the average of the final 99 trials. Each search terminates when either the target vertex is reached or all the vertices are visited.

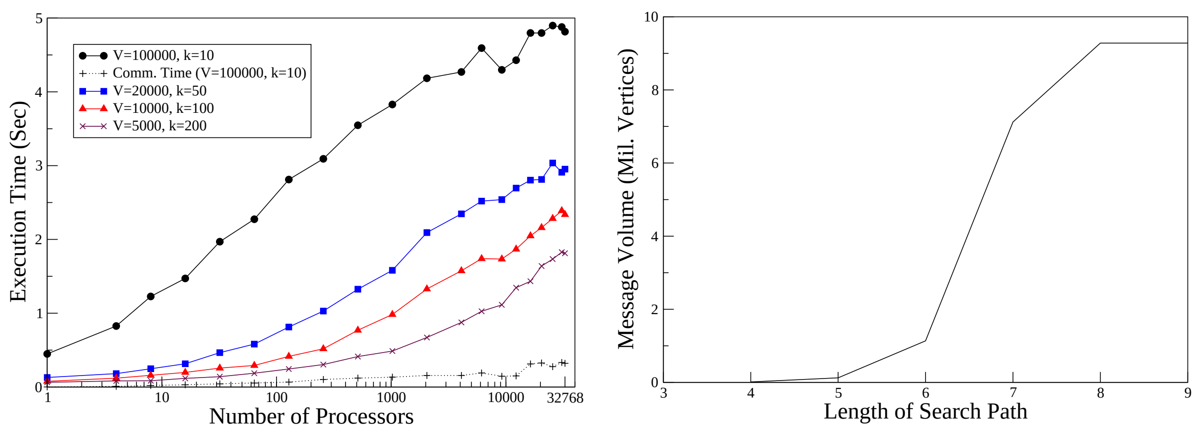
First, the actual message lengths in distributed BFS for 1-D and 2-D partitionings have been measured on the MCR Cluster and are plotted in Figure 2. The results are qualitatively similar to the bounds derived in Section 3 and plotted in Figure 1. For example, the crossover points of 1-D and 2-D curves in Figures 1.a and 2 are around the average degree of 25, beyond which 2-D curve grows in a slower rate than 1-D curve.

We measured the scalability of the proposed BFS algorithm in weak scaling experiments on a 32768-node BlueGene/L system and present the results in Figure 3. In a weak scaling study, we increase the global problem size as the number of processors increases. Therefore, the size of local problem assigned to each processor remains constant. The local problem size used in these experiments is 100000 vertices and the average degree of the graphs varies from 10 to 200.

The scalability of our BFS scheme is clearly demonstrated in Figure 3.a. The largest

Algorithm 1 Distributed Breadth-First Expansion with 1-D Partitioning

Initialize $L_{v_s}(v) = \begin{cases} 0, & v = v_s \\ \infty, & \text{otherwise} \end{cases}$
for $l = 0$ to ∞ **do**
  $F \leftarrow \{v \mid L_{v_s}(v) = l\}$, the set of local vertices with level l
  if $F = \emptyset$ for all processors **then**
    Terminate main loop
  end if
  $N \leftarrow$ {neighbors of vertices in F (not necessarily local)}
  for all processors q **do**
    $N_q \leftarrow$ {vertices in N owned by processor q}
    Send N_q to processor q
    Receive $\bar{N}_q$ from processor q
  end for
  $\bar{N} \leftarrow \bigcup_q \bar{N}_q$ (The $\bar{N}_q$ may overlap)
  for $v \in \bar{N}$ and $L_{v_s}(v) = \infty$ **do**
    $L_{v_s}(v) \leftarrow l + 1$
  end for
end for



(a) Mean search time

(b) Message volume per level

Figure 3: Weak scaling results of the distributed BFS on 32768-node BlueGene/L system. V denotes the number of local vertices ($\frac{n}{P}$) and k denotes the average degree. The number of processors is shown in a logarithmic scale.

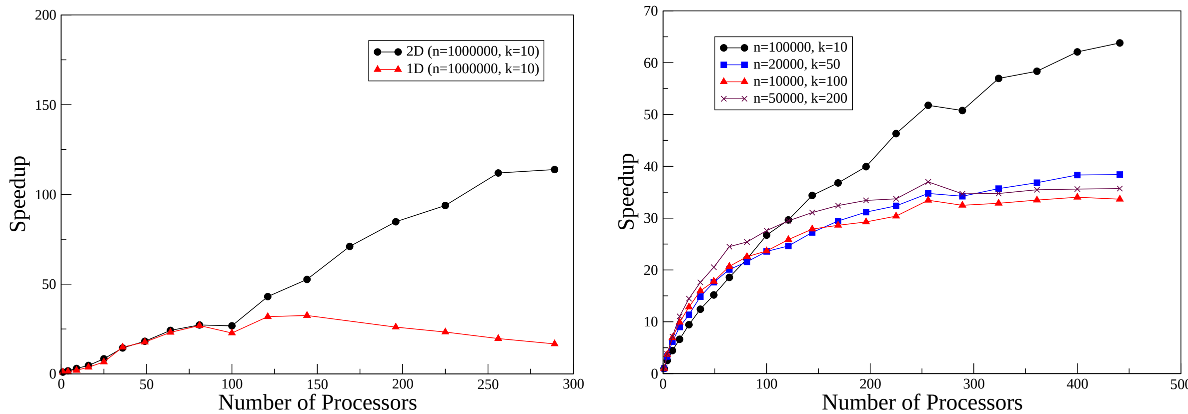
Algorithm 2 Distributed Breadth-First Expansion with 2-D Partitioning

Initialize $L_{v_s}(v) = \begin{cases} 0, & v = v_s \\ \infty, & \text{otherwise} \end{cases}$
for $l = 0$ to ∞ **do**
  $F \leftarrow \{v \mid L_{v_s}(v) = l\}$, the set of local vertices with level l
  if $F = \emptyset$ for all processors **then**
    Terminate main loop
  end if
  for all processors q in this processor-column **do**
    Send F to processor q
    Receive $\bar{F}_q$ from processor q (The $\bar{F}_q$ are disjoint)
  end for
  $\bar{F} \leftarrow \bigcup_q \bar{F}_q$
  $N \leftarrow \left\{\text{neighbors of vertices in } \bar{F} \text{ using edge lists on this processor}\right\}$
  for all processors q in this processor-row **do**
    $N_q \leftarrow \{\text{vertices in } N \text{ owned by processor } q\}$
    Send N_q to processor q
    Receive $\bar{N}_q$ from processor q
  end for
  $\bar{N} \leftarrow \bigcup_q \bar{N}_q$ (The $\bar{N}_q$ may overlap)
  for $v \in \bar{N}$ and $L_{v_s}(v) = \infty$ **do**
    $L_{v_s}(v) \leftarrow l + 1$
  end for
end for

graph used in this study has 3.2 billion vertices and 32 billion edges. To the best of our knowledge, this is the largest graph ever explored by a distributed graph search algorithm. The high scalability of our scheme can be attributed to the fact that the length of message buffers used in our algorithm does not increase as the size of graphs grows. Figure 3.a reveals that the communication time is very small compared to the computation time (in the case with local problem size of 100000 vertices and the average degree of 10). This indicates that our algorithm is highly memory-intensive as it involves very little computation. Profiling of the code has confirmed that it spends most of its time in a hashing function that is invoked to process the received vertices. The communication time for other graphs with different degrees are also very small and omitted in the figure for clarity.

It is shown in Figure 3.a that the execution time curves increase in proportion to $\log P$, where P is the number of processors, and it is confirmed by a regression analysis. Part of the reason for the logarithmic scaling factor is that the search time for a graph is dependent on the length of path between the source and destination vertices, and the path length is bounded by the diameter of the graph, which is $O(\log n)$ for a random graph with n vertices [3]. That is, n increases proportionally as P increases in weak scaling study, and therefore the diameter of the graph (and the search time) increases in proportion to $\log P$. The performance of the BFS algorithm improves as the average degree, k, increases. This is obvious, because as the degree of vertices increases the length of a path being searched decreases, and hence the search time decreases. Note, however, that for larger average degree, the execution time increases faster than $\log(n)$.

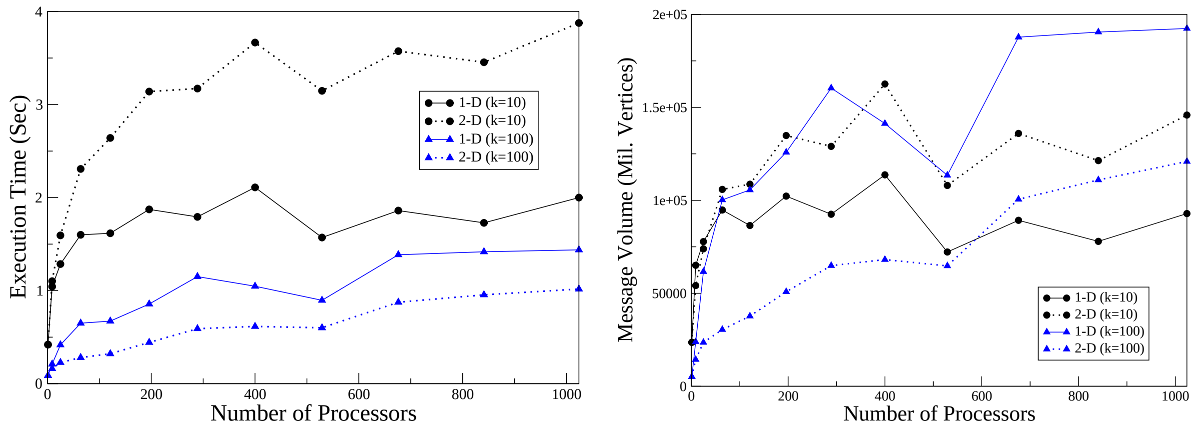
The Figure 3.b shows the total volume of messages received by our BFS algorithm as a function of the number of levels used in the search. These results are for a small graph with 12 million vertices and 120 million edges. It can be clearly seen in the figure that the message volume increases quickly as the path length increases until the path length reaches the diameter of the graph.



(a) MCR (b) BlueGene/L

Figure 4: Strong scaling results of the distributed BFS on MCR Cluster and BlueGene/L systems. In the figures, n denotes the number of (global) vertices in a given graph and k denotes the average vertex degree.

Figure 4 shows the results from a strong scaling test. In contrast to weak scaling, the global problem size (the number of vertices in a given graph) remains constant while the number of processors increases. Therefore, the number of local vertices assigned to each processor decreases as the number of processors increases. In Figure 4.a which presents the strong scaling results measured on MCR for a graph with 1 million vertices and 10 million edges, the scaling is worse for 1-D partitioning, because 1-D partitioning behaves worse for small local problem sizes. The decline in speedup signals the point where increased communication overhead outweighs any performance gain obtained by increased parallelism. The strong scaling results for 2-D partitioning on BlueGene/L are presented in Figure 4.b. Here graphs with different number of vertices and edges are used as indicated in the figure. As expected, the speedup curves grows in proportion to $\sqrt{P}$ for small P, where P is the number of processors. For larger P, the speedup tapers off as the local problem size becomes very small and the communication overhead becomes dominant.

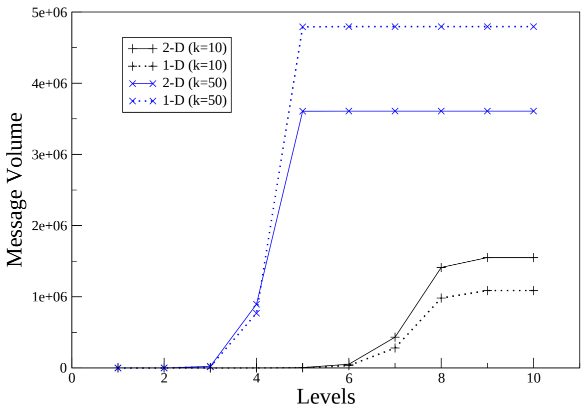


(a) Execution time (b) Message volume

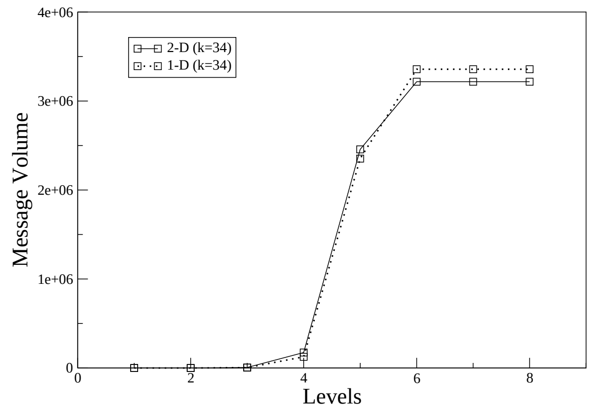
Figure 5: Weak scaling test on IBM BlueGene/L for 1-D and 2-D partitionings. The local problem sizes of 100000 and 10000 vertices are used with the average degrees of 10 and 100, respectively.

Figure 5 compares weak scaling test results for 1-D and 2-D partitionings on BlueGene/L. Graphs with the average degree of 10 and 100 are considered in the experiments. The numbers of local vertices assigned to each processor are 100000 and 10000 for average degrees of 10 and 100 respectively to make the number of edges in the graphs to be the same for the given number of processors, independent of the average degree of the graphs. The search time and the total message volume are measured. Figures 5.a and 5.b clearly show that there is a strong correlation between the search time and the message volume. As explained earlier, this is because our algorithm is highly memory-intensive and the number of memory accesses is governed by the lengths of messages sent and received by a processor during search. Furthermore, the average degree of a graph has great impact on the performance of 1-D and 2-D partitionings, since it determines the lengths of messages transmitted by each processor to expand a frontier. Figure 5.a shows that 1-D partitioning performs better than 2-D partitioning when the average degree is small (10 in this case), and 2-D partitioning outperforms 1-D partitioning for large average degree. These findings coincide with results shown in Figure 1. For a low-degree graph, the lengths of messages sent and received by a processor increase slowly as the search progresses. In other words, the breadth-first tree grows slowly for low-degree graphs. This favors 1-D partitioning, since the total volume of messages transmitted by a processor per level-expansion is small with the 1-D partitioning. On the contrary, the 2-D partitioning would require more messages to expand a frontier, as it involves two communication steps. For a high-degree graph, on the other hand, the breadth-first tree grows rapidly. Therefore, the total volume of messages transmitted by a processor per level-expansion is large with 1-D partitioning, since with the 1-D partitioning each processor is likely to exchange large messages with all the other processors to expand a frontier. With 2-D partitioning, however, a small set of processors perform the level-expansion by exchanging smaller messages, compared with 1-D partitioning, with one another. Although 2-D partitioning involves two communications per each level-expansion, this results in reduction in total message volume.

The effect of the average degree of a graph on the performance of partitioning schemes is more evident in Figure 6, which plots the volume of messages received by a processor at each level-expansion of the search. level in a search. Graphs with 40 million vertices with average degrees of 10 and 50 are analyzed in this study. We have used an unreachable target vertex in the search to capture the worst-case behavior of the partitioning schemes. Figure 6.a shows that the message volume increases more slowly with 1-D partitioning than with 2-D partitioning for a low-degree graph. For a high-degree graph, 2-D partitioning generates less messages than 1-D partitioning. In fact, using the equations to calculate the message lengths for 1-D and 2-D partitionings given in Section 3, we can determine the average degree of a Poisson random graph for which 1-D and 2-D partitionings show the similar performance. We have computed such an average degree for a graph with 40 million vertices and compared the performance of 1-D and 2-D partitionings with the graph in Figure 6.b. The figure shows



(a) $k = 10$ and $k = 50$



(b) $k = 34$

Figure 6: Message volume as a function of level in a search. Graphs with 40 million vertices are used. In (b), the value of k is derived from an equation, $n \cdot \gamma(\frac{n}{P}) \cdot \frac{P-1}{P} = 2 \cdot \frac{n}{P} \cdot \gamma(\frac{n}{\sqrt{P}}) \cdot (\sqrt{P}-1)$, where $P = 400$ and $n = 40000000$.

that 1-D and 2-D partitionings exhibit almost identical performance when the average degree is 34.

We have implemented the fold communication as a reduce-scatter operation, where the reduction is set-union operation. The effectiveness of this fold communication, called *union-fold*, is demonstrated on BlueGene/L and the results are presented in Figure 7. We have used the redundancy ratio as performance metric in this experiment. The redundancy ratio is defined as the ratio of duplicate vertices eliminated by the union-fold operation to the total number of vertices received a processor. Obviously, more redundant vertices can be eliminated by the union-fold operation for the graph with the higher degree (100). It is shown that the union-fold operation can save as much as 80% of vertices received by each processor. Although the proposed union operation requires copying of received messages incurring additional overhead, it reduces the total number of vertices to be processed by each processor and ultimately improves overall performance by reducing memory accessing

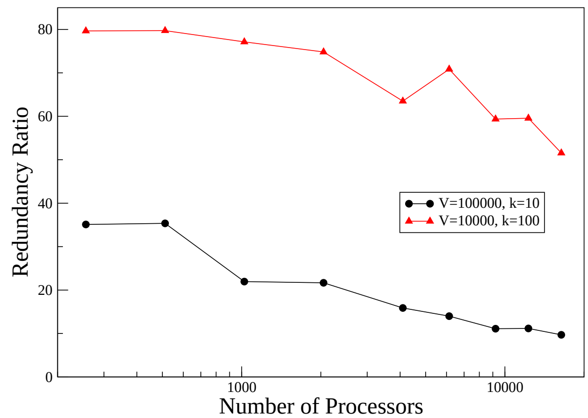


Figure 7: Performance of the union-fold operation on BlueGene/L. V denotes the number of local vertices ($\frac{n}{P}$) and k denotes the average degree. The number of processors is shown in a logarithmic scale.

time. The redundancy ratio declines for both graphs, however, as the number of processors increases. It has been shown in Figure 3.b that the message length increases exponentially as search expands its frontiers until the path length approaches the diameter of the graph, after which the message length remains constant. Therefore, total message length received by each processor should be almost constant independent of the number of processors in a weak scaling run, since the diameter of the graph increases very slowly especially for large graphs. This implies that the number of duplicate vertices in received messages should be constant as well. However, in our union-fold operation each processor receives more messages as the number of processors increases, because it passes the messages using ring communications. This is why the redundancy ratio declines as more processors are used.

5 Conclusions

This paper presents a distributed parallel BFS algorithm with 2-D partitioning. We have demonstrated large-scale distributed BFS with Poisson random graphs with as many as 3 billion vertices and 30 billion edges in this research. To the best of our knowledge, this is the largest graph ever explored by a distributed graph search algorithm. We have analyzed 1-D and 2-D partitionings theoretically and experimentally. We have found that our BFS algorithm is highly memory-intensive and 2-D partitionings are to be preferred when the average degree of the graph is large. Further, for both 1-D and 2-D partitionings of Poisson random graphs, the memory usage per processor is scalable.

Future work should address graphs besides Poisson random graphs, e.g., graphs with large clustering coefficient and scale-free graphs, which are graphs with a few vertices of very large degree. Graphs with large diameter should also be investigated for completeness, although a level-synchronized algorithm may perform poorly for these cases.

Acknowledgments

The authors are pleased to thank Doruk Bozdağ, Ümit Çatalyürek, and Tina Eliassi-Rad for helpful discussions on this work. This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

References

[1] M. Barthélemy, E. Chow, and T. Eliassi-Rad. Knowledge representation issues in semantic graphs for relationship detection. In *2005 AAAI Spring Symposium on AI Technologies for Homeland Security*, 2005.

[2] Blue Gene/L. http://cmg-rr.llnl.gov/asci/platforms/bluegenel.

[3] B. Bollobás. The diameter of random graphs. *Trans. American Mathematical Society*, 267:41–52, 1981.

[4] Ü. V. Çatalyürek. *Hypergraph Models for Sparse Matrix Partitioning and Reordering.* PhD thesis, Bilkent University, 1999.

[5] Ü. V. Çatalyürek and C. Aykanat. Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. on Parallel and Distributed Systems*, 10(7):673–693, 1999.

[6] Ü. V. Çatalyürek and C. Aykanat. A hypergraph-partitioning approach for coarse-grain decomposition. In *ACM/IEEE SC2001*, Denver, CO, November 2001.

[7] C. Faloutsos, K. McCurley, and A. Tomkins. Fast discovery of connection subgraphs. In *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 118–127, Seattle, WA, USA, 2004. ACM Press.

[8] Geoffrey Fox, Mark Johnson, Steve Otto, John Salmon, and David Walker. *Solving Problems on Concurrent Processors.* Prentice-Hall, Inc., 1988.

[9] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. Technical report, Sandia National Laboratories, 1993.

[10] Bruce Hendrickson, Robert Leland, and Steve Plimpton. An efficient parallel algorithm for partitioning irregular graphs. *Int. Journal of High Speed Computing*, 7(1):73–88, 1995.

[11] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal of Scientific Computing*, 20(1), 1999.

[12] J. G. Lewis, D. G. Payne, and R. A. van de Geijn. Matrix-vector multiplication and conjugate gradient algorithms on distributed memory computers. In *Proceedings of the Scalable High Performance Computing Conference*, pages 542–550, 1994.

[13] J. G. Lewis and R. A. van de Geijn. Distributed memory matrix-vector multiplication and conjugate gradient algorithms. In *Proceedings of Supercomputing'93*, pages 484–492, Portland, OR, November 1993.

[14] Multiprogrammatic Capability Cluster (MCR). http://www.llnl.gov/linux/mcr.

[15] M. E. J. Newman. From the cover: The structure of scientific collaboration networks. *Proceedings of the National Academy of Sciences*, 98:404–409, 2001.

[16] Andrew T. Ogielski and William Aiello. Sparse matrix computations on parallel processor arrays. *SIAM Journal on Scientific Computing*, 14(3):519–530, 1993.

[17] Jesper Larsson Träff. An experimental comparison of two distributed single-source shortest path algorithms. *Parallel Comput.*, 21(9):1505–1532, 1995.